Computer System Networking and Telecommunications

2022 Volume 1, Issue 1

ISSN: 2630-4635

A Countermeasure Against a Whitelist-Based Access Control Bypass Attack Using Dynamic DLL Injection Scheme

Dae-Youb Kim*

Department of Information Security, The University of Suwon, Gyeonggi Province, Republic of Korea

Corresponding author: Dae-Youb Kim, daeyoub69@suwon.ac.kr

Abstract

Traditional malware detection technologies collect known malicious programs and analyze their characteristics. Then, a blacklist is made based on the malicious characteristics detected. The user's program is then checked based on the blacklist to determine the presence of malware. However, such an approach can only detect known malicious programs but not unknown ones. In addition, since such detection technologies generally monitor all programs in the system in real time, they might affect the system's performance. In order to solve such problems, various methods have been proposed to analyze the major behaviors of malicious programs and how to respond to them. Ransomware is designed to access and encrypt the user's file. Therefore, a new approach is to produce a whitelist of programs installed in the user's system and to only allow the programs listed on the whitelist to access the user's files. However, even with this approach, attackers can still launch a dynamic link library (DLL) injection attack on a regular program registered on the whitelist. Hence, this paper proposes a method to respond effectively to DLL injection attacks.

Keywords

Malware
Ransomware
Blacklist
Whitelist
DLL injection

1. Introduction

Traditionally, the technologies for detecting malware involve collecting malicious programs, analyzing them at the code level, and extracting their characteristics. Malware is detected by comparing and analyzing programs installed or stored in the user's system based on the characteristics extracted. This technique utilizes a traditional access control technology called blacklist, which is a list of inaccessible objects, and is very effective in detecting known malware in the system.

However, blacklist-based malware detection techniques cannot detect new and unknown malware or malware variants. In addition, these techniques involve scanning all files stored in the system, which can degrade the system's performance [1].

To address these issues, several methods have been suggested focusing on monitoring program behavior rather than its code characteristics to determine if it is malicious. For example, several studies have attempted to detect unknown malware using machine learning techniques [2,3].

An alternative approach is to implement access control measures on processes that serve a particular objective, like ransomware, to regulate programs engaged in malicious activities. In the case of ransomware, it is characterized by accessing and encrypting certain types of files stored in the target process. Therefore, some techniques have been proposed to control the behavior of ransomware by applying access control technology to specific files or file directories [4,5].

To design an efficient detection program considering the behavioral characteristics of ransomware that accesses specific files stored on the target system, the following two points can be considered.

(1) Scanning all programs on a system for ransomware in real-time is highly inefficient. Thus, it is essential to restrict the scope by specifying the types and quantity of programs to be scanned.

(2) There are many different types of malicious programs such as ransomware, and the number is growing rapidly. On the other hand, the average user typically works with a limited variety of file types, such as documents, images, and multimedia, and the number of programs they use to access these files is usually not extensive.

In light of these considerations, it is more effective to have a whitelist of programs that are normally allowed to access certain types of files, and to allow only those programs to access them, rather than a blacklist based on an analysis of the collected ransomware. To this end, when establishing file access policies within an operating system, an "All Access Deny" access control policy can be implemented by default and a whitelist can be created to control the scope of access granted. In other words, a list (whitelist) of the types of files that need to be protected on the system and the programs that are used to access those files normally can be created, and programs other than those included in the whitelist are not allowed to access the files [5-7]. This approach to access control has also been proposed as a default security system for the Windows operating system. In the case of controlled folder access (CFA), which has been proposed to counter ransomware on Windows operating systems, access to specific folders is controlled to prevent access by illegal processes such as ransomware [8].

However, whitelisting access control also has its limitations. A dynamic link library (DLL) injection attack involves injecting malicious code into a whitelisted program, allowing unauthorized access to files stored on the system. In this case, the program targeted by the DLL injection attack is whitelisted and is therefore allowed to access files stored on the system, resulting in illegal file access. In fact, it has been shown that CFA, which was introduced as a ransomware countermeasure in Windows operating systems, can be disabled by DLL injection attacks [9].

Therefore, in order to detect and respond to ransomware using whitelist-based access control technology, a countermeasure against bypass attacks using DLL injection attacks is essential.

In this paper, we propose a technique for monitoring DLL injection attacks on whitelist-based access control solutions and controlling DLL injection attacks when a program that has previously been granted file access is attacked. By utilizing the proposed technique, whitelist-based access control technology can be implemented more securely. In addition, we applied the proposed technique to evaluate its potential for ransomware control and its performance [5].

In **Section 2**, the whitelist-based ransomware detection techniques are reviewed. **Section 3** describes the utilization DLL injection attack and the countermeasures proposed. **Section 4** describes the results of the test code applying the proposed technique, and **Section 5** is the conclusion of the research.

2. Whitelist-based ransomware detection

To compensate for the shortcomings of existing blacklist-based ransomware detection solutions and to enhance response capabilities, a whitelist-based ransomware detection solution has been proposed. **Figure 1** illustrates the concept of the whitelist-based ransomware detection and response solution proposed by Kim *et al* [5]. In the Windows operating system, a program's file access request is typically handled by the I/O manager, which generates an I/O request packet (IRP) and forwards it to the file system filter driver (Step B). The file system filter driver processes the IRP and delivers the processed result to the program that requested file access through the I/O manager (Steps C and D). The technique proposed is designed so that the IRPs generated by the I/O manager are obtained by the file usage monitor (FUM) before the file system filter driver and access permissions are determined. FUM can be implemented by utilizing Windows minifilters [10]. The procedure of the proposed solution is as follows [5]:

(A) When a program requests file access, the I/O manager generates an IRP to handle the request.

(B1) The FUM obtains that IRP ahead of the file system filter driver.

(B2) The FUM analyzes the IRP information and passes the information about the program that is trying to access the file to the file access control manager (FAM).

(B3) The FAM checks whether the process information is recorded in the whitelist it manages.

(B4) The FAM allows access if the process is included in the whitelist, and denies access otherwise.

(B5) The FAM communicates its decision to the FUM.

(B6) If the FUM receives a file access denial, it deletes the IRP and terminates processing.

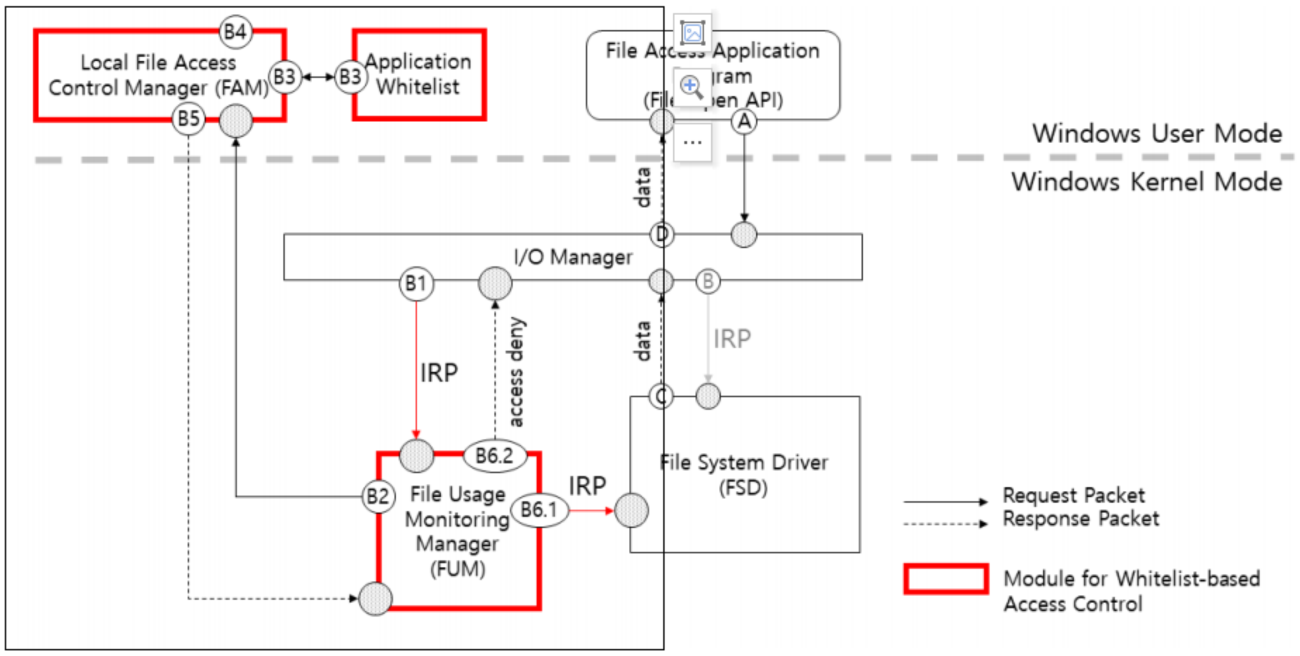


Figure 1. Whitelist-based Ransomware Prevention System [5]

Otherwise, it passes the IRP to the file system filter driver so that the program can perform the remaining procedures to access the file normally.

3. DLL injection attacks and countermeasures

3.1. DLL injection attack

Libraries used in software program development are mainly divided into static libraries and dynamic link libraries (DLLs). Static libraries are distributed as part of the executable during the program's executable configuration phase, while DLLs are designed to load libraries into memory as needed during program execution or to share libraries already loaded by other processes.

A DLL injection attack is a technique that exploits the dynamic linking nature of DLLs to illegally insert and execute attack code into a running process. DLL injection attacks involve injecting malicious code into legitimate processes to gain unauthorized access or evade detection by security solutions designed to identify malicious processes. There are two main ways to perform DLL injection attacks: static DLL injection and dynamic DLL injection.

(1) Static DLL injection attack

When a program legitimately utilizes DLLs, the operating system analyzes the program's portable executable (PE) file as it is being loaded into memory to obtain the list of DLL files it needs. This DLL file information is obtained by analyzing the import directory table (IDT) of the PE file. Static DLL injection attacks can be carried out in two ways.

(i) Modifying the PE file: creating an abnormal DLL file and adding the DLL file information to the IDT of the PE file.

(ii) DLL file modification: creating an abnormal DLL file with the same name as the normal DLL file. In this case, the abnormal DLL file is designed to include the functions of the normal DLL.

Static DLL injection technology modifies the PE file or DLL file of a legitimately installed process in the system, so it can be detected by utilizing the code signature of the legitimate file.

(2) Dynamic DLL injection attack

A dynamic DLL injection attack involves injecting an additional attack DLL file into a process (or program) running on the system, and the attack is executed automatically by implementing the attack code in the DLLMain function, which is automatically executed when the DLL file is loaded into memory. A typical way to perform a dynamic DLL injection attack within the Windows operating system is to exploit the CreateRemoteThread function to cause the LoadLibraryA function to load an attack DLL file into the target process. A dynamic DLL injection attack using the CreateRemoteThread function is performed according to the following procedure [11]. To perform a dynamic DLL injection attack, the attacker first creates two files.

(i) The malicious DLL file to be injected into the target process: the actual attack code is designed to be called from the DLLMain function of the DLL file.

(ii) An attack program (DLL injector) file to perform the DLL injection: The attacker runs the DLL injector to remotely inject the previously created attack DLL file into the target process.

In this case, the DLL injector is designed to perform the attack according to the following procedure.

(i) The execution privilege of the injector is set to SE_DEBUG_NAME. This allows the injector to have debugging privileges and memory access.

(ii) The process ID (PID) of the target process is obtained. The PID can be used to request a handler for the target process from the system.

(iii) Memory is allocated to the target process and the path to the location where the attack DLL file is stored in the allocated memory.

(iv) The system is searched for the starting address of the LoadLibraryA function currently in memory. Since the LoadLibraryA function is one of the Win32 APIs and also uses kernel32.dll, which is one of the basic system DLLs of the Windows operating system, the memory address of the function is determined when kernel32.dll is loaded into memory after the operating system boots. Due to the nature of DLL file operations, all subsequent processes that use the function will use the same function address.

(v) The handler of the target process, the starting address of the LoadLibraryA function, and the path to the offensive DLL file are specified as input parameters to the CreateRemoteThread function so that the DLL file is loaded in the target process. If the offensive DLL file is successfully mounted in the target process, the DLLMain function of the offensive DLL file is automatically called.

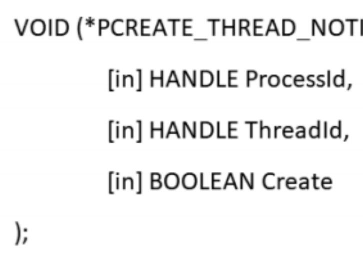
Dynamic DLL injection attacks do not require modification of the PE file of the target process, nor do they change DLL files already in use by processes on the system. However, when access control techniques such as whitelisting are used to control a process' access to files, dynamic DLL injection attacks can bypass or defeat them.

3.2. Detecting dynamic DLL injection attacks

The most popular technique for dynamic DLL injection attacks is remote thread creation. Therefore, it is possible to detect DLL injection attacks if thread-related events occurring within the system can be monitored, so suspicious threads can be identified. In fact, Windows operating systems can monitor events such as process or thread creation/termination and image loading in real-time. Therefore, it is possible to utilize these features to monitor dynamic DLL injection attack attempts.

(1) Monitoring thread events

CreateThreadNotifyRoutine refers to a routine that is called when a new thread is created or an existing thread is terminated. This routine is executed not only when the Main Thread of the process is created/terminated, but also when the CreateThread function or CreateRemoteThread function is called to create a new thread. These callback functions can be registered using the PsSetCreateThreadNotifyRoutine function provided by Windows. This thread notification callback function is designed to use the parameter NotifyRoutine of type PCREATE_THREAD_NOTIFY_ROUTINE to specify the pointer value of the function to be registered to be called when a thread event occurs. The primitive form of this callback function is shown in **Figure 2**. The thread ID (TID) and PID supplied by the operating system through the callback function are the ID of the thread in which the event occurred and the ID of the process in which the thread is mounted, respectively.



```
VOID (*PCREATE_THREAD_NOTIFY_ROUTINE)(
        [in] HANDLE ProcessId,
        [in] HANDLE ThreadId,
        [in] BOOLEAN Create
);
```

Figure 2. Thread notification callback function type

Within this callback function, you can call the PsGetCurrentProcessId function and PsGetCurrentThreadId function to get the PID and TID that created the thread, but it is important to note that the PID and TID obtained are not the IDs of the process/thread that called the function.

(2) Suspicious thread detection criteria

Ko *et al.* proposed to monitor suspicious processes and thread creation events by utilizing the thread event monitoring technique described above [4]. This technique is proposed to monitor and respond to malware processes or threads that are commonly exploited by malware.

When a DLL injection attack is attempted in a normal process using the CreateRemoteThread function, the PID and TID are set according to the caller and callee as shown in **Table 1**.

In **Table 1**, column 1 of the ID column refers to the process and TID values passed by the thread notification callback functions, respectively. Columns 2 and 3 of the ID column refer to the process and TID values obtained by calling the PsGetCurrentProcessId and PsGetCurrentThreadId functions to check the creator information of the thread where the event occurred within the thread notification callback function, respectively.

Case 1 means that process A (PA) creates process B (PB), and Case 2 means that PA mounts a thread (TA) in PA for its own use. Both Case 1 and Case 2 are considered normal behavior.

Case 3 is when a PA mounts a thread on a PB. As shown in **Table 1**, in this case, the PID and cPID values have different ID values, which means that the process where the thread is mounted and the process that created the thread are different processes. The same thing happens the CreateRemoteThread function is used to remotely inject a DLL. However, Case 1 also shows similar results to Case 3, but in Case 1, the main thread of PB is created, which can be distinguished by checking the number of threads created in the process.

Therefore, if Case 3 is detected within the thread notification callback function, it is considered as a suspicious thread creation event and the thread information is managed by adding it to the suspicious thread list.

However, since the technique proposed by Ko *et al.* [4] only determines whether there is a DLL injection attack, it cannot determine whether the injected DLL is used in a ransomware attack.

In this paper, we improved the proposed thread monitoring technique so that it applies to ransomware control solutions and verified its performance by applying it to Kim *et al.*'s solution [5]. Therefore, we propose to further monitor the file access of suspicious threads and use it for file access control. In addition, we proposed a control procedure that can be integrated into the existing whitelist-based access control technology.

3.3. Countering whitelist-based access control bypass attacks

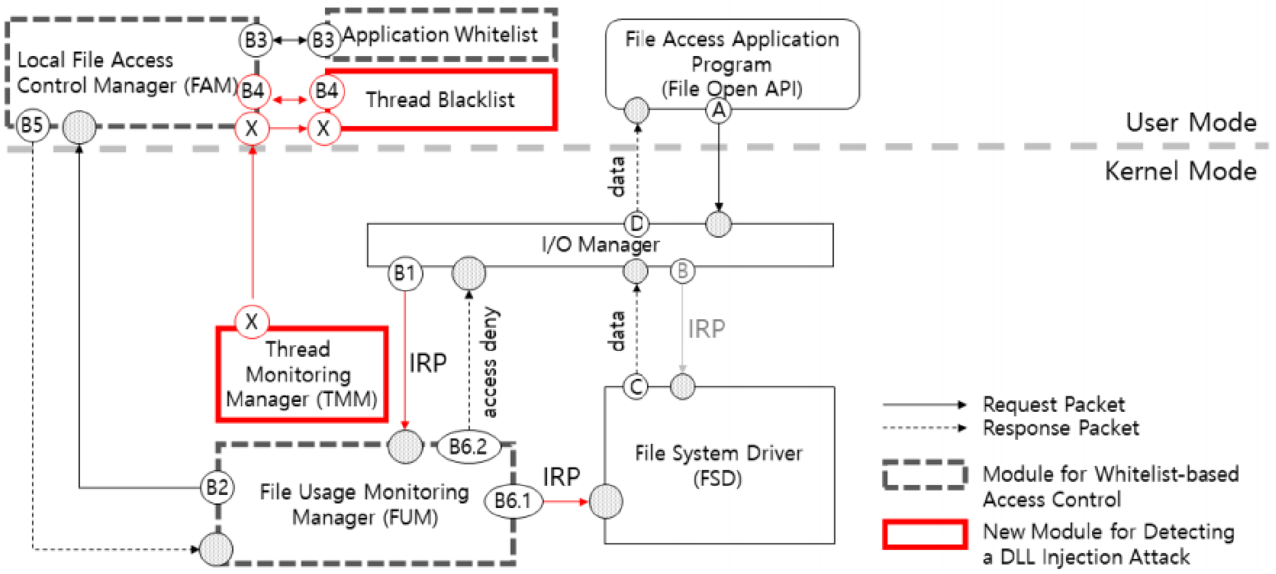
To respond to DLL injection attacks, we improved the whitelist-based ransomware response monitoring technology described above.

(1) The PsGetCurrentThreadId function is called within the I/O callback function of the minifilter used by the whitelist-based ransomware response monitoring to obtain the creator ID value of the thread trying to access the file. The constructor ID value of the thread is compared with the ID values of the suspicious thread list created earlier.

(2) If the same ID is found in the list of suspicious threads as the creator ID of the thread, the thread is assumed to have been created by a

Table 1. Process/Thread ID

ID \ Case		Case 1	Case 2	Case 3
		PA→TA	PA→TB	
1	PID	PID[B]	PID[A]	PID[B]
	TID	TID[B]	TID[A]	TID[B]
2	cPID	PID[A]	PID[A]	PID[A]
3	cTID	TID[A]	TID[A]	TID[A]



Local File Access Control Manager (FAM)
B3
B3
Application Whitelist
B4
B4
Thread Blacklist
B5
X
X
File Access Application Program (File Open API)
A
User Mode
Kernel Mode
data
D
I/O Manager
B1
B
data
IRP
C
X
Thread Monitoring Manager (TMM)
IRP
access deny
B6.2
File System Driver (FSD)
B2
File Usage Monitoring Manager (FUM)
B6.1
IRP
Request Packet
Response Packet
Module for Whitelist-based Access Control
New Module for Detecting a DLL Injection Attack

Figure 3. A countermeasure procedure against a bypass attack using DLL injection skill

DLL injection attack, and access to the file is denied. **Figure 3** illustrates the procedure for detecting and responding to whitelist-based access control bypass attacks.

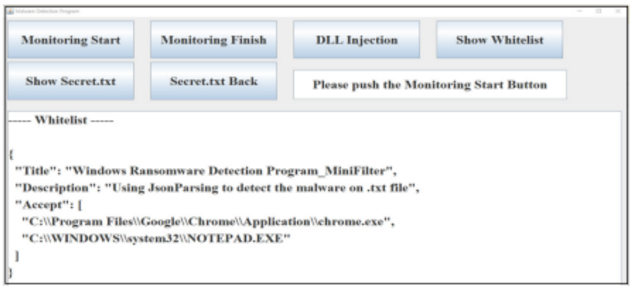
In **Figure 1**, it was proposed that the file access control manager compares and analyses the process information collected in step B2 and step B3 in step B4 to determine the access rights of the process trying to access the file and notify the file usage monitoring manager of the judgment result to handle whether the process requesting access to the file is allowed or not. In the scheme proposed in this paper, in step X, the thread monitoring manager (TMM) sends the suspicious thread information to the file access control manager, and the manager adds the information to the thread blacklist. Step X is performed independently of the steps in the existing file access procedure.

(3) When a program attempts to access a file, the whitelisting access control procedure is repeated from step B1 to step B3 based on the information of the program's file access request process [5]. If the whitelist information determines that the requesting process has legitimate access rights, then in step B4, the file access control manager analyzes the thread blacklist information to determine whether the thread attempting to access the file is included in the blacklist. At this point, the IRP generated to process the program's file access request is analyzed to determine if the IRP information contains any threads that are included in the thread blacklist. If the thread of the process recorded in the IRP is blacklisted, the file usage monitoring administrator is notified to deny file access.

4. Implementation results and performance evaluation

4.1. Implementation results

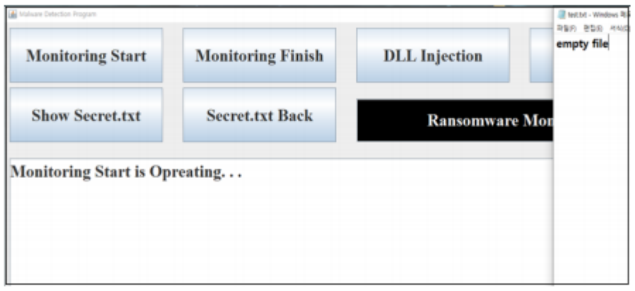
Figure 4 shows the implementation results of the proposed countermeasures against DLL injection attacks, which shows that when a DLL injection attack is performed against a whitelisted program, the program becomes a host for ransomware and encrypts files: (a) shows the ransomware detection program proposed in Kim *et al.*'s paper [5]. The detection program includes chrome.exe and notepad.exe in the whitelist that defines the access permissions of the .txt



Please push the Monitoring Start Button

```
----- Whitelist -----

{
 "Title": "Windows Ransomware Detection Program_MiniFilter",
 "Description": "Using JsonParsing to detect the malware on .txt file",
 "Accept": [
  "C:\\Program Files\\Google\\Chrome\\Application\\chrome.exe",
  "C:\\WINDOWS\\system32\\NOTEPAD.EXE"
 ]
}
```

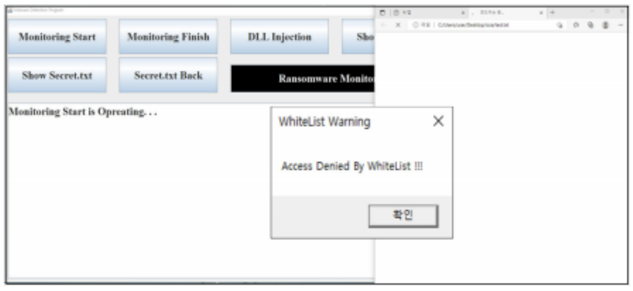
(a) Test Program UI and Whitelist



Monitoring Start is Opreating. . .

empty file

(b) Application with Access Right

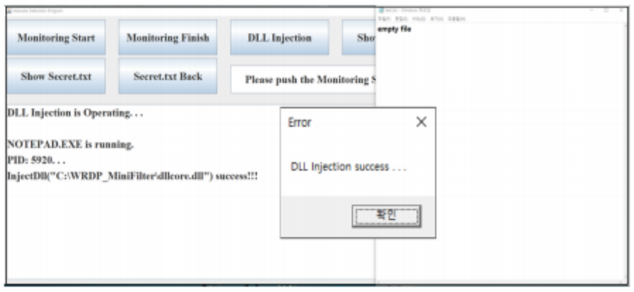


Monitoring Start is Opreating. . .

WhiteList Warning

Access Denied By WhiteList !!!

(c) Application without Access Right



Please push the Monitoring S

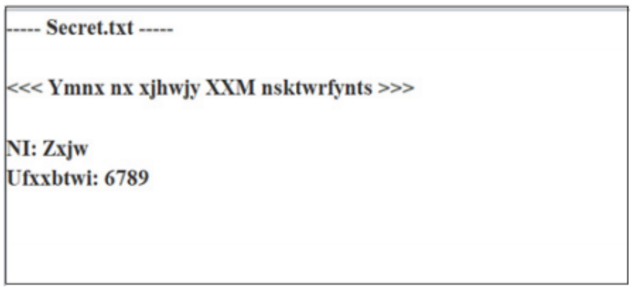
```
DLL Injection is Operating. . .

NOTEPAD.EXE is running.
PID: 5920. . .
InjectDll("C:\WRDP_MiniFilter\dllcore.dll") success!!!
```

Error

DLL Injection success . . .

empty file

(d) DLL Injection Attack

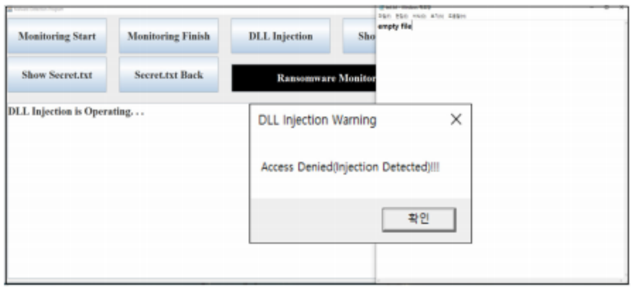


----- Secret.txt -----

<<< Ymnx nx xjhwjy XXM nsktwrfynts >>>

NI: Zxjw
Ufxxbtwi: 6789

(e) The Result of DLL Injection Attack



DLL Injection is Operating. . .

DLL Injection Warning

Access Denied(Injection Detected)!!!

empty file

(f) A Countermeasure for DLL Injection

Figure 4. Implementation

file. (b) shows the result of accessing the test file using notepad.exe. (c) shows the result of trying to access the test file using a program other than the whitelisted program, and being denied access. In (d), the attack DLL has been successfully injected into notepad.exe. (e) shows that the file has been encrypted by the attack DLL injected into notepad.exe. (f) shows the result when the notepad.exe with the malicious DLL tries to access the file, and the access request is denied.

4.2. Performance analysis

Unlike existing approaches to detect and respond to ransomware, Kim *et al.*'s solution [5] is designed to control the behavior of ransomware by applying a whitelist-based access control technique to control access to files by all processes except authorized programs. Therefore, not only the ransomware selected from the ransomware database, but the newly implemented ransomware for testing was also measured to have a malicious activity success rate of 0%.

Since this paper is developed to improve the performance of the whitelist-based ransomware detection solution proposed by Ko *et al.*, we controlled the file access of all processes that have no record in the whitelist. In addition, even for whitelisted programs, we simulated the case where a DLL injection attack that

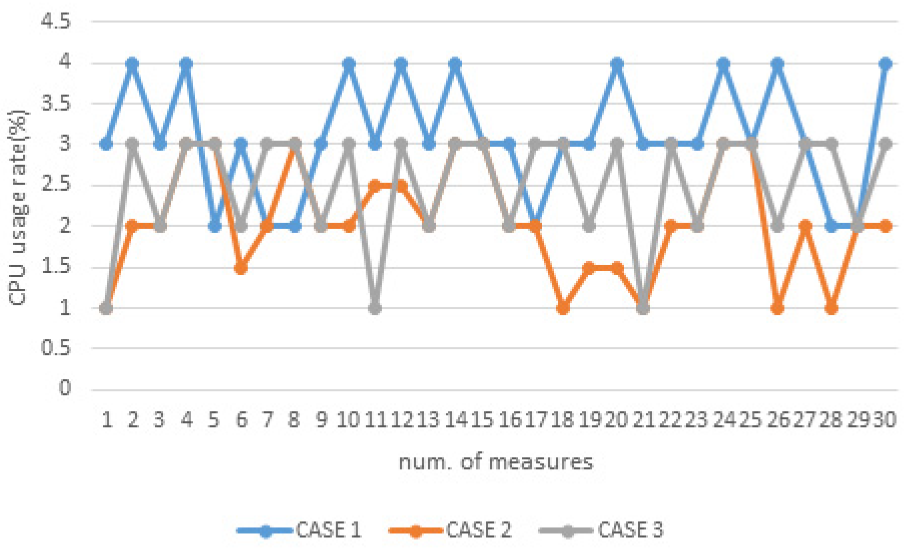


Figure 5. Evaluation (CPU usage rate) result

exploits remote thread creation is performed against the program, and the file access that exploits DLL injection technology was also effectively controlled.

Figure 5 shows the results of the system performance analysis. For the performance analysis, the CPU occupancy of the system was measured 30 times assuming three cases.

(1) Case 1 was measured after running Notepad and a malware detection program on the system.

(2) Case 2 is the result of running Notepad and a ransomware detection solution implemented with the technology proposed in this paper.

(3) Case 3 was measured after running the ransomware detection solution implemented in WordPad and the technique proposed in this paper and performing a DLL injection attack.

The measurement environment was as follows: CPU (Intel i7-10700F), RAM (32GB), OS (Windows 10, 64 bit). As shown in the results for Case 2 in **Figure 5**, it is clear that even with the addition of the ability to watch for suspicious threads, the average CPU occupancy was 2.05%, which was more efficient than the 3.1% average occupancy of the general-purpose malware analysis solution in Case 1. Also, when identifying and responding to actual DLL injection attacks. As for Case 3, the average occupancy was 2.5%, which was more efficient than the 3.1% in Case 1.

The implementation performance of the added/improved module is as follows: Under the same conditions as in Case 3.

(1) When a DLL injection attack was attempted on Notepad, the time taken by the TMM in **Figure 3** to detect it and update the suspicious thread list was 110 ns on average.

(2) The time taken by FUM and FAM (**Figure 3)** to determine the program's file access rights based on the suspicious thread list and whitelist, and to control the malicious behavior according to the access control policy was 170 ns on average.

5. Conclusion

In Kim *et al.*'s paper [5] and Microsoft [8], a new ransomware countermeasure solution that uses access control technology to control file access of suspected ransomware processes was proposed. These technologies were able to control ransomware by establishing access control policies based on whitelists and controlling file access of all processes other than legitimate authorized processes. However, it was pointed out that DLL injection technology can bypass these control policies.

To improve this vulnerability, we incorporated DLL injection attack monitoring technology into a whitelist-based ransomware solution to enhance safety. To improve its efficiency, only DLL injection attacks on specific applications were monitored and thread blacklists were managed. In addition, IRPs and thread blacklists were analyzed to process file access requests of programs in conjunction with a whitelist-based access control solution to determine the attack status.

By applying the proposed technology to Kim *et al.*'s solution [5], we improved the security of a whitelist-based ransomware solution by blocking programs suspected of executing DLL injection attacks from accessing files. We also measured the system resource consumption due to the added DLL injection attack countermeasure and found that it was more efficient than existing malware detection solutions.

Funding

The paper was supported by a research grant from the University of Suwon in 2021.

Disclosure statement

The author declares no conflict of interest

References

[1] Chakkaravarthy S, Sangeetha D, Vaidehi V, 2019, A Survey on Malware Analysis and Mitigation Techniques. Computer Science Review, 32: 1–23. https://doi.org/10.1016/j.cosrev.2019.01.002

[2] Gibert D, Mateu C, Planes J, 2020, The Rise of Machine Learning for Detection and Classification of Malware: Research Developments, Trends and Challenges. Journal of Network and Computer Applications, 153(1): 102526. https://doi.org/10.1016/j.jnca.2019.102526

[3] Khammas B, 2020, Ransomware Detection using Random Forest Technique, ICT Express, vol.6, no.4,. https://doi.org/10.1016/j.icte.2020.11.001

[4] Ko BS, Choi WH, Jeong DJ, 2020, A Study on the Tracking and Blocking of Malicious Actors through Thread Based Monitoring, Korea Institute of Information Security and Cryptology, 30(1): 75–86. https://doi.org/10.13089/JKIISC.2020.30.1.75

[5] Kim D, Lee J, 2020, Blacklist vs. Whitelist-Based Ransomware Solutions. IEEE Consumer Electronics Magazine, 9(3): 22–28. https://doi.org/10.1109/MCE.2019.2956192

[6] McIntosh T, Kayes A, Chen Y, et al., 2021, Ransomware Mitigation in the Modern Era: A Comprehensive Review, Research Challenges, and Future Directions. Computer Science ACM Computing Surveys (CSUR), 7(9): 1–36. https://doi.org/10.1145/3479393

[7] Kim S, Hwang I, Kim D, 2021, A Study on Creation of Secure Storage Area and Access Control to Protect Data from Unspecified Threats. Journal of the Society of Disaster Information, 17(4): 897–903. https://doi.org/10.15683/kosdi.2021.12.31.897

[8] Enable Controlled Folder Access, n.d., https://learn.microsoft.com/en-us/microsoft-365/security/defender-endpoint/

enable-controlled-folders?view=o365-worldwide

[9] Abrams L, 2018, Windows 10 Ransomware Protection Bypassed Using DLL Injection, BLEEPINGCOMPUTER, https://www.bleepingcomputer.com/news/security/windows-10-ransomware-protection-bypassed-using-dll-injection/

[10] Filter Manager and Minifilter Driver Architecture, n.d., https://docs.microsoft.com/ko-kr/windows-hardware/drivers/ifs/filtermanager-concepts